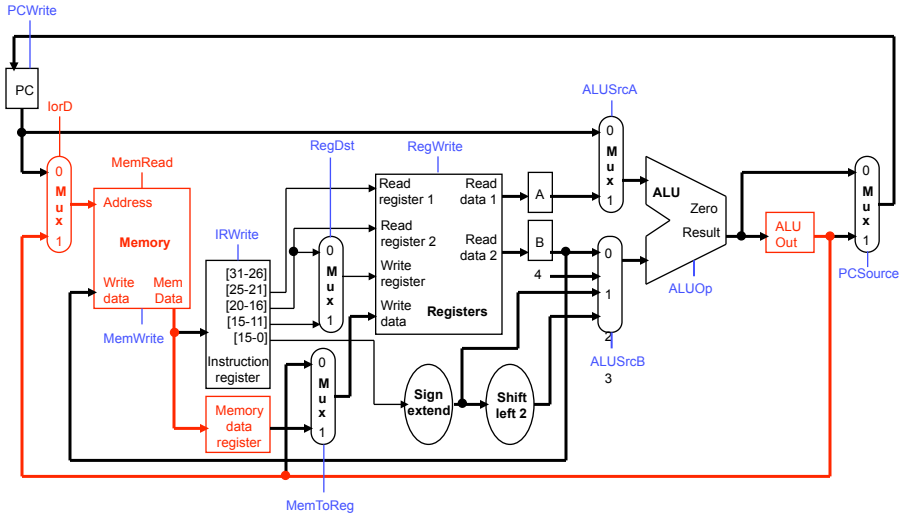
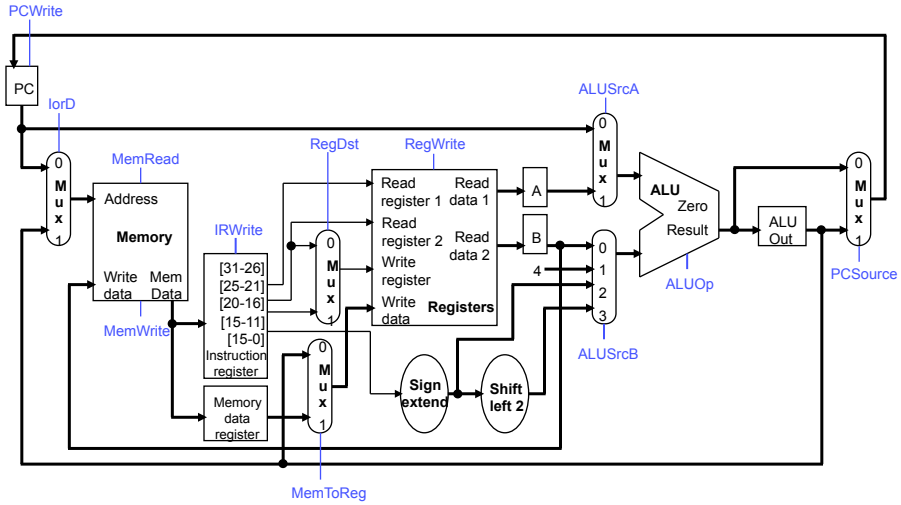
Review Multicycle: What is Happening



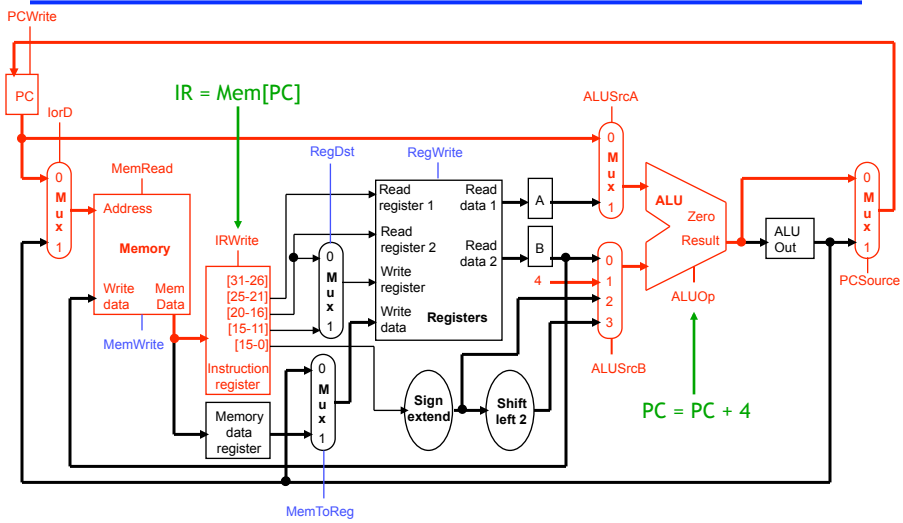
1

Controlling The Multicycle Design



2

Stage 1: Instruction fetch & PC increment

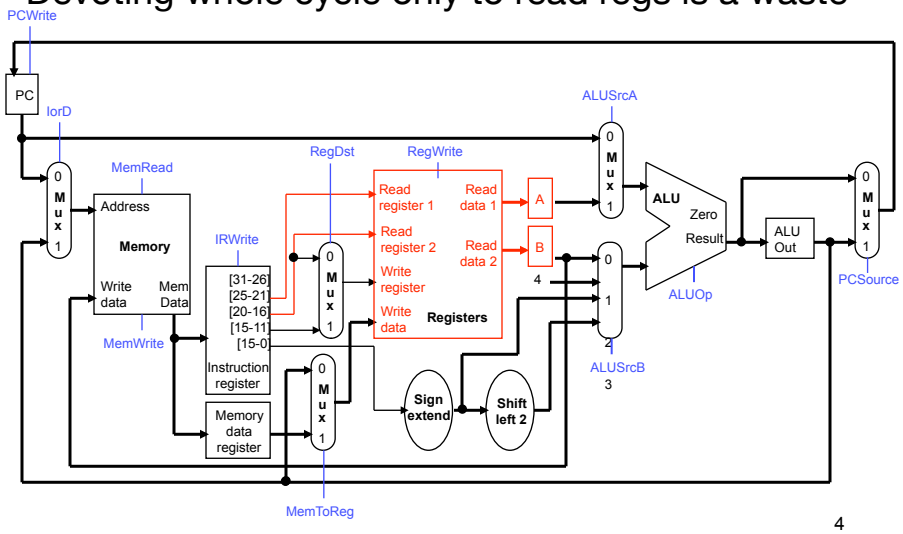


Controls: PCWrite, IorD, MemRead, IRWrite, ALUSrcA==0, ALUSrcB==1, ALUOp==add, PCSource==0

3

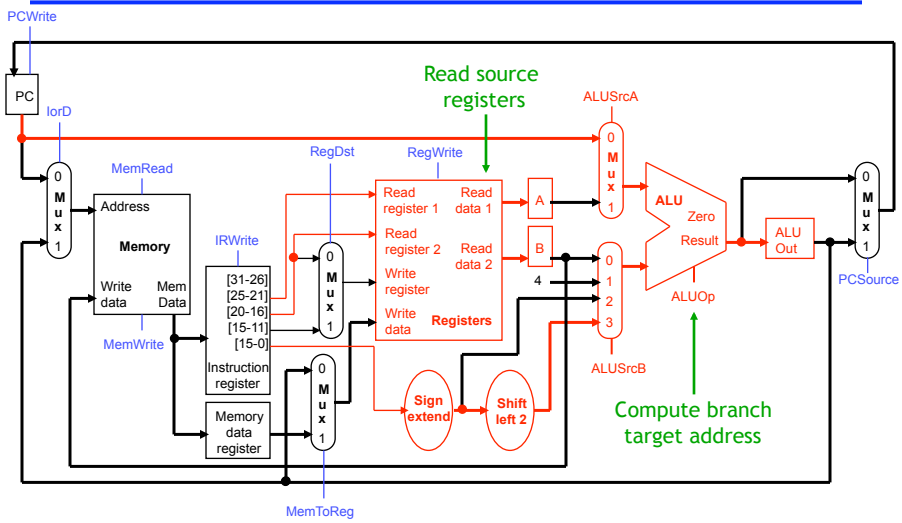
Register File Read

Devoting whole cycle only to read regs is a waste



4

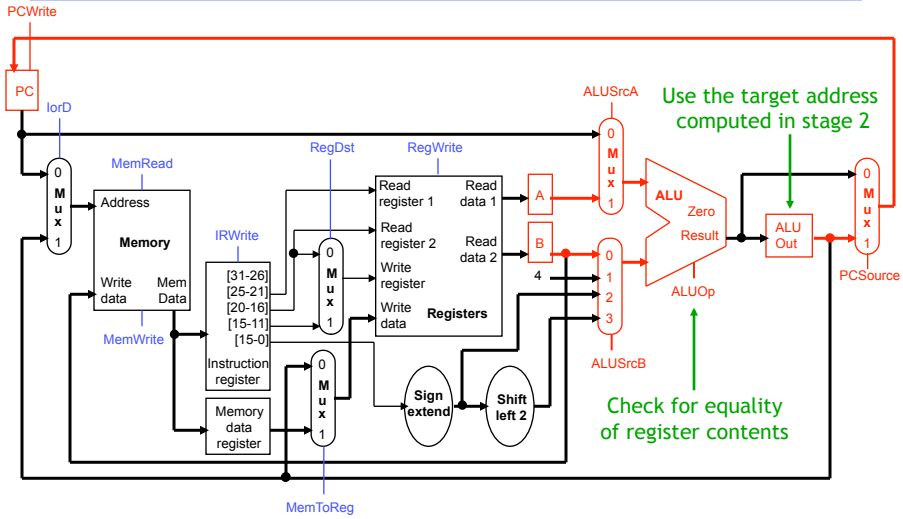
Stage 2: Reg fetch & branch target



Controls: ALUSrcA==0, ALUSrcB==3, ALUOp==add

5

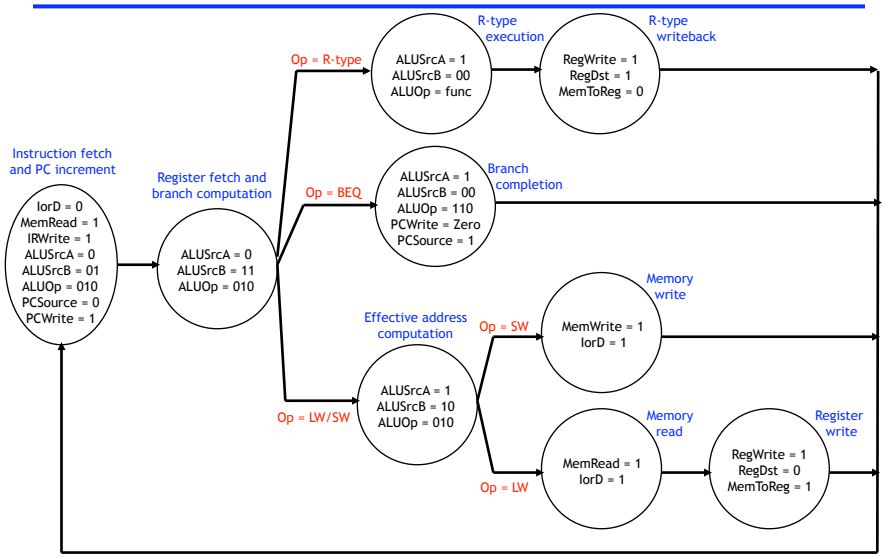
Stage 3 (beq): Branch completion



Controls: ALUSrcA==1, ALUSrcB==0, ALUOp==sub, PCSource==1, PCWrite==1

6

Finite-state machine for the control unit



All instruction are the same for stages 1 and 2

7

Comparing instruction execution times

In the single-cycle datapath, each instruction needs an entire clock cycle, or 8ns, to execute

With the multicycle CPU, different instructions need different numbers of clock cycles

- A branch needs 3 cycles, or 3 x 2ns = 6ns
- Arithmetic and sw instructions each require 4 cycles, or 8ns
- Finally, a lw takes 5 cycles, or 10ns

We can make some observations about performance already

- Loads take *longer* with this multicycle implementation, while all other instructions are faster than before.
- So if our program doesn't have too many loads, then we should see an increase in performance.

8

The gcc example

Let's assume the gcc instruction mix

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

In a single-cycle datapath, all instructions take 8ns

The average execution time for an instruction on the multicycle processor works out to 8.06ns:

(48% x 8ns) + (22% x 10ns) + (11% x 8ns) + (19% x 6ns)
= 3.84 + 2.2 + .88 + 1.14 = 8.06ns

The multicycle implementation is actually slightly slower

9

Reconsider Memory's Role

Memory is 50ns, implying single-cycle = 104ns implying a 9.6MHz clock rate

For multi-cycle w/cache, let the processor stall on a cache miss

- Keep 2ns cycle time or 500MHz clock rate
- Instruction execution for GCC 8.06 ns

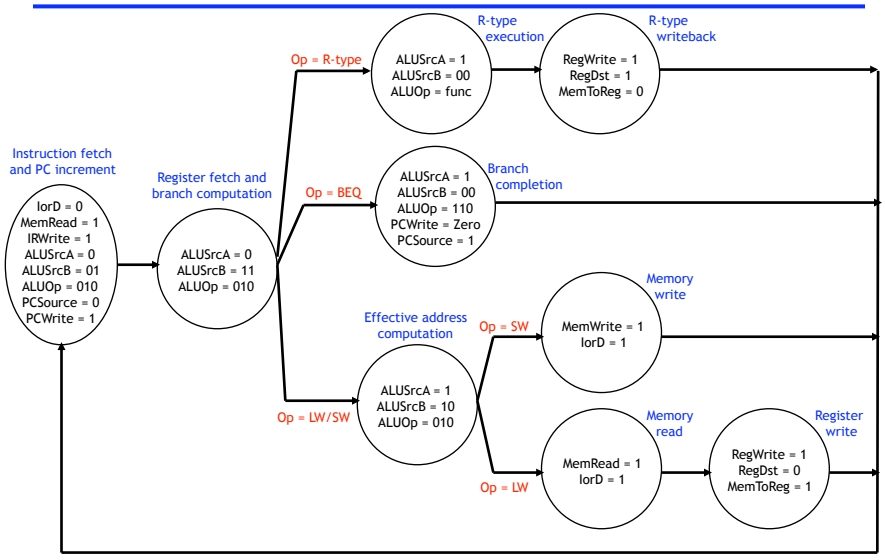
Consider executing 10^9 instructions w/ 10^6 memory references: 50ns * 10^6 = 5*10^7 ns

single-cycle = 104 seconds for total of 104.05 sec

multi-cycle = memory time + instruction execution time = 0.05 + 8.06 seconds for total of 8.11 sec

10

Return:Finite-state machine for control



11

Recall: Implementing the FSM

FSM can be translated into a state table; first 2 states:

Current State	Input (Op)	Next State	Output (Control signals)												
			PC Write	IorD	MemRead	Mem Write	IR Write	Reg Dst	MemTo Reg	Reg Write	ALU SrcA	ALU SrcB	ALU Op	PC Source
Instr Fetch	X	Reg Fetch	1	0	1	0	1	X	X	0	0	01	010	0
Reg Fetch	BEQ	Branch compl	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	R-type	R-type execute	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	LW/SW	Compute eff addr	0	X	0	0	0	X	X	0	0	11	010	X

You can implement this the hard way – you don't want to do this

- Represent the current state using flip-flops or a register.
- Find equations for the next state and (control signal) outputs in terms of the current state and input (instruction word).

Or you can use the easy way.

- Stick the whole state table into a memory, like a ROM
- This would be much easier, since you don't have to derive equations

12

Motivation for microprogramming

Think of the control unit's state diagram as a program

- Each state represents a "command," or a set of control signals that tells the datapath what to do
- Several commands are executed sequentially
- "Branches" may be taken depending on the instruction opcode
- The state machine "loops" by returning to the initial state

We could invent a special language for the control unit

- We could devise a more readable, higher-level notation rather than dealing directly with binary control signals and state transitions
- We would design control units by writing "programs" in this language
- We would depend on a hardware or software translator to convert our programs into a circuit for the control unit

13

A good notation is very useful

Instead of specifying the exact binary values for each control signal, we will define a symbolic notation that's easier to work with

As a simple example, we might replace ALUSrcB = 01 with ALUSrcB = 4, meaning the constant 4

We can also create symbols that *combine* several control signals together. Instead of

IorD = 0
MemRead = 1
IRWrite = 1

it would be nicer to just say something like

Read PC

14

Microinstructions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

For the MIPS multicycle we could define microinstructions with eight fields.

- These fields will be filled in symbolically, instead of in binary
- They determine all the control signals for the datapath. There are only 8 fields because some of them specify more than one of the 12 actual control signals
- A microinstruction corresponds to one execution stage, or one cycle

You can see that in each microinstruction, we can do something with the ALU, register file, memory, and program counter units

15

Specifying ALU operations

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

ALU control selects the ALU operation

- Add indicates addition for memory offsets or PC increments
- Sub performs source register comparisons for "beq"
- Func denotes the execution of R-type instructions

SRC1 is either PC or A, for the ALU's first operand

SRC2, the second ALU operand, can be one of four different values

- B for R-type instructions and branch comparisons
- The constant 4 to increment the PC
- Extend, the sign-extended constant field for mem refs
- Extshift, sign-extended, shifted constant for branch targets

These correspond to the ALUOp, ALUSrcA and ALUSrcB control signals, except we use names like "Add" and not actual bits like "010."

16

Specifying register and memory actions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

Register control selects a register file action
- Read to read from registers "rs" and "rt" of the instruction word
- Write ALU writes ALUOut into destination register "rd"
- Write MDR saves MDR into destination register "rt"

Memory chooses the memory unit's action
- Read PC reads an instruction from address PC into IR
- Read ALU reads data from address ALUOut into MDR
- Write ALU writes register B to address memory ALUOut

17

Specifying PC actions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

PCWrite control determines what happens to the PC.
- ALU sets PC to ALUOut, used in incrementing the PC.
- ALU-Zero writes ALUOut to PC only if the ALU's Zero condition is true. This is used to complete a branch instruction.

Next determines the next microinstruction to be executed.
- Seq causes the next microinstruction to be executed.
- Fetch returns to the initial instruction fetch stage.
- Dispatch i is similar to a "switch" or "case" statement; it branches depending on the actual instruction word.

18

Microprogramming the first stage

Below are two lines of microcode to implement the first two multicycle execution stages, instruction fetch and register fetch

The first line, labeled Fetch, involves several actions

- Read from memory address PC
- Use ALU to compute PC + 4, and return it to the PC
- Continue on to the next sequential microinstruction

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1

19

The second stage

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1

The second line implements register fetch stage

- Read registers rs and rt from the register file
- Pre-compute PC + (sign-extend(IR[15-0]) << 2) for branches
- Determine the next microinstruction based on the opcode of the current MIPS program instruction

```
switch (opcode) {
    case 4:   goto BEQ1;
    case 0:   goto Rtype1;
    case 43:
    case 35:  goto Mem1;
}
```

20

Completing a beq instruction

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
BEQ1	Sub	A	B			ALU-Zero	Fetch

Control would transfer to this microinstruction if the opcode was "beq"

- Compute A-B, to set the ALU's Zero bit if A=B
- Update PC with ALUOut (which contains the branch target from the previous cycle) if Zero is set
- The beq is completed, so fetch the next instruction

The 1 in the label BEQ1 reminds us that we came here via the first branch point ("dispatch table 1"), from the second execution stage

21

Completing an arithmetic instruction

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Rtype1	func	A	B				Seq
				Write ALU			Fetch

When the opcode indicates an R-type instruction…

- The first cycle performs an operation on registers A and B, based on the MIPS instruction's func field
- The next stage writes the ALU output to register "rd" from the MIPS instruction word

We can then go back to the Fetch microinstruction, to fetch and execute the next MIPS instruction

22

Completing data transfer instructions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Mem1	Add	A	Extend				Dispatch 2
SW2					Write ALU		Fetch
LW2					Read ALU		Seq
				Write MDR			Fetch

For both sw, lw instructions, we should first compute the effective memory address, A + sign-extend(IR[15-0])

Another dispatch or branch distinguishes between stores and loads

- For sw, we store data (from B) to the effective memory address
- For lw we copy data from the effective memory address to register rt

In either case, we continue on to Fetch when done

23

Microprogramming vs. programming

Microinstructions correspond to control signals

- They describe what is done in a single clock cycle
- These are the most basic operations available in a processor

Microprograms implement higher-level MIPS instructions

- MIPS assembly language instructions are comparatively complex, each possibly requiring multiple clock cycles to execute
- But each complex MIPS instruction can be implemented with several simpler microinstructions

24

Similarities with assembly language

Microcode is intended to make control unit design easier

- We defined symbols like Read PC to replace binary control signals
- A translator converts microinstructions into a real control unit
- The translation is straightforward, because each microinstruction corresponds to one set of control values

This sounds similar to MIPS assembly language!

- We use mnemonics like lw instead of binary opcodes like 100011
- MIPS programs must be *assembled* to produce real machine code
- Each MIPS instruction corresponds to a 32-bit instruction word

25

Managing complexity

It looks like all we've done is devise a new notation that makes it easier to specify control signals

And that's exactly right! The issue is managing complexity

- Control units are probably the most challenging part of CPU design
- Large instruction sets require large state machines with many states, branches and outputs
- Control units for multicycle processors are difficult to create and maintain

Applying programming ideas to hardware design is a useful technique

26

Cases when microprogramming is bad

One disadvantage of microprograms is that looking up control signals in a ROM can be *slower* than generating them from simplified circuits

Sometimes complex instructions implemented in hardware are *slower* than equivalent assembly programs written using simpler instructions

- Complex instructions are usually very general, so they can be used more often. But this also means they can't be optimized for specific operands or situations
- Some microprograms just aren't written very efficiently. But since they're built into the CPU, people are stuck with them (at least until the next processor upgrade)

27

How microcode is used today

Modern CISC processors (like x86) use a combination of hardwired logic and microcode to balance design effort with performance

- Control for many simple instructions can be implemented in hardwired which can be faster than reading a microcode ROM
- Less-used or very complex instructions are microprogrammed to make the design easier and more flexible (floats, divide)

In this way, designers respect the "first law of performance"

- Make the common case fast!

28